22 | 函数：怎么结合多返回值进行错误处理？

2021-12-01 Tony Bai

《Tony Bai · Go语言第一课》 课程介绍 >

讲述：Tony Bai
时长 20:29 大小 18.77M

你好，我是 Tony Bai。

上一节课，我们开始了 Go 函数的学习，对 Go 语言中的函数已经有了基础的了解。那么，今天这节课，我们要再进一步，学习怎么做好函数设计。

在上节课的函数声明部分，我们提到，多返回值是 Go 语言函数，区别于其他主流静态编程语言中函数的一个重要特点。同时，它也是 Go 语言设计者建构 Go 语言错误处理机制的基础，而错误处理设计也是做函数设计的一个重要环节。

所以今天这节课，我们将会从 Go 语言的错误处理机制入手，围绕 Go 语言错误处理机制的原理、Go 错误处理的常见策略，来学习一下如何结合函数的多返回值机制进行错误处理的设计。

这会让你建立起 Go 编码的统一错误处理思维，写出更健壮的、让你自己更有信心的 Go 代码。

要想做好错误处理设计，我们首先要先来了解 Go 语言错误设计的基本思路与原理。

Go 语言是如何进行错误处理的？

采用什么错误处理方式，其实是一门编程语言在设计早期就要确定下来的基本机制，它在很大程度上影响着编程语言的语法形式、语言实现的难易程度，以及语言后续的演进方向。

我们前面已经多次提到，Go 语言继承了“先祖”C 语言的很多语法特性，在错误处理机制上也不例外，Go 语言错误处理机制也是在 C 语言错误处理机制基础上的再创新。

那么这里，我们依然从源头讲起，先看看前辈 C 语言的错误处理机制。在 C 语言中，我们通常用一个类型为整型的函数返回值作为错误状态标识，函数调用者会基于值比较的方式，对这一代表错误状态的返回值进行检视。通常，这个返回值为 0，就代表函数调用成功；如果这个返回值是其它值，那就代表函数调用出现错误。也就是说，函数调用者需要根据这个返回值代表的错误状态，来决定后续执行哪条错误处理路径上的代码。

C 语言的这种简单的、**基于错误值比较**的错误处理机制有什么优点呢？

首先，它让每个开发人员必须显式地去关注和处理每个错误，经过显式错误处理的代码会更健壮，也会让开发人员对这些代码更有信心。

另外，你也可以发现，这些错误就是普通的值，所以我们不需要用额外的语言机制去处理它们，我们只需利用已有的语言机制，像处理其他普通类型值一样的去处理错误就可以了，这也让代码更容易调试，更容易针对每个错误处理的决策分支进行测试覆盖。C 语言错误处理机制的这种简单与显式结合的特征，和 Go 语言设计哲学十分契合，于是 Go 语言设计者决定继承 C 语言这种错误处理机制。

不过 C 语言这种错误处理机制也有一些弊端。□比如，由于 C 语言中的函数最多仅支持一个返回值，很多开发者会把这单一的返回值“一值多用”。什么意思呢？就是说，一个返回值，不仅要承载函数要返回给调用者的信息，又要承载函数调用的最终错误状态。比如 C 标准库中的`fprintf`函数的返回值就承载了两种含义。在正常情况下，它的返回值表示

输出到 FILE 流中的字符数量，但如果出现错误，这个返回值就变成了一个负数，代表具体的错误值：

复制代码

```
// stdio.h
int fprintf(FILE * restrict stream, const char * restrict format, ...);
```

特别是当返回值为其他类型，比如字符串的时候，我们还很难将它与错误状态融合到一起。这个时候，很多 C 开发人员要么使用输出参数，承载要返回给调用者的信息，要么自定义一个包含返回信息与错误状态的结构体，作为返回值类型。大家做法不一，就很难形成统一的错误处理策略。

为了避免这种情况，Go 函数增加了**多返回值机制**，来支持错误状态与返回信息的分离，并建议开发者把要返回给调用者的信息和错误状态标识，分别放在不同的返回值中。

我们继续以上面 C 语言中的 fprintf 函数为例，Go 标准库中有一个和功能等同的 `fmt.Fprintf`的函数，这个函数就是使用一个独立的表示错误状态的返回值（如下面代码中的 err），解决了 fprintf 函数中错误状态值与返回信息耦合在一起的问题：

复制代码

```
// fmt包
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error)
```

我们看到，在 fmt.Fprintf 中，返回值 n 用来表示写入 io.Writer 中的字节个数，返回值 err 表示这个函数调用的最终状态，如果成功，err 值就为 nil，不成功就为特定的错误值。

另外我们还可以看到，fmt.Fprintf 函数声明中代表错误状态的变量 err 的类型，并不是一个传统使用的整数类型，而是用了一个名为 error 的类型。

虽然，在 Go 语言中，我们依然可以像传统的 C 语言那样，用一个整型值来表示错误状态，但 **Go 语言惯用法**，是使用 error 这个接口类型表示错误，并且按惯例，我们通常将 error 类型返回值放在返回值列表的末尾，就像 fmt.Fprintf 函数声明中那样。

那么 error 接口类型究竟如何表示错误？我们又该如何构造一个满足 error 接口类型的错误值呢？我们继续向下看。

error 类型与错误值构造

error 接口是 Go 原生内置的类型，它的定义如下：

复制代码

```
// $GOROOT/src/builtin/builtin.go
type interface error {
    Error() string
}
```

任何实现了 error 的 Error 方法的类型的实例，都可以作为错误值赋值给 error 接口变量。那这里，问题就来了：**难道为了构造一个错误值，我们还需要自定义一个新类型来实现 error 接口吗**？

Go 语言的设计者显然也想到了这一点，他们在标准库中提供了两种方便 Go 开发者构造错误值的方法： `errors.New`和`fmt.Errorf`。使用这两种方法，我们可以轻松构造出一个满足 error 接口的错误值，就像下面代码这样：

复制代码

```
err := errors.New("your first demo error")
errWithCtx = fmt.Errorf("index %d is out of bounds", i)
```

这两种方法实际上返回的是同一个实现了 error 接口的类型的实例，这个未导出的类型就是`errors.errorString`，它的定义是这样的：

复制代码

```
// $GOROOT/src/errors/errors.go

type errorString struct {
    s string
}

func (e *errorString) Error() string {
    return e.s
}
```

大多数情况下，使用这两种方法构建的错误值就可以满足我们的需求了。但我们也要看到，虽然这两种构建错误值的方法很方便，但它们给错误处理者提供的错误上下文（Error Context）只限于以字符串形式呈现的信息，也就是 Error 方法返回的信息。

但在一些场景下，错误处理者需要从错误值中提取出更多信息，帮助他选择错误处理路径，显然这两种方法就不能满足了。这个时候，我们可以自定义错误类型来满足这一需求。比如：标准库中的 net 包就定义了一种携带额外错误上下文的错误类型：

复制代码

```
// $GOROOT/src/net/net.go
type OpError struct {
    Op string
    Net string
    Source Addr
    Addr Addr
    Err error
}
```

这样，错误处理者就可以根据这个类型的错误值提供的额外上下文信息，比如 Op、Net、Source 等，做出错误处理路径的选择，比如下面标准库中的代码：

复制代码

```
// $GOROOT/src/net/http/server.go
func isCommonNetReadError(err error) bool {
    if err == io.EOF {
        return true
    }
    if neterr, ok := err.(net.Error); ok && neterr.Timeout() {
        return true
    }
    if oe, ok := err.(*net.OpError); ok && oe.Op == "read" {
        return true
    }
    return false
}
```

我们看到，上面这段代码利用类型断言，判断 error 类型变量 err 的动态类型是否为 *net.OpError 或 net.Error。如果 err 的动态类型是 *net.OpError，那么类型断言就会返

回这个动态类型的值（存储在 oe 中），代码就可以通过判断它的 Op 字段是否为"read"来判断它是否为 CommonNetRead 类型的错误。

不过这里，你不用过多了解类型断言（Type Assertion）到底是什么，你只需要知道通过类型断言，我们可以判断接口类型的动态类型，以及获取它动态类型的值接可以了。后面我们在讲解接口类型的时候还会再细讲。

那么，使用 error 类型，而不是传统意义上的整型或其他类型作为错误类型，有什么好处呢？至少有这三点好处：

第一点：统一了错误类型。

如果不同开发者的代码、不同项目中的代码，甚至标准库中的代码，都统一以 error 接口变量的形式呈现错误类型，就能在提升代码可读性的同时，还更容易形成统一的错误处理策略。这个我们下面会细讲。

第二点：错误是值。

我们构造的错误都是值，也就是说，即便赋值给 error 这个接口类型变量，我们也可以像整型值那样对错误做“==”和“!=”的逻辑比较，函数调用者检视错误时的体验保持不变。

第三点：易扩展，支持自定义错误上下文。

虽然错误以 error 接口变量的形式统一呈现，但我们很容易通过自定义错误类型来扩展我们的错误上下文，就像前面的 Go 标准库的 OpError 类型那样。

error 接口是错误值的提供者与错误值的检视者之间的契约。error 接口的实现者负责提供错误上下文，供负责错误处理的代码使用。这种错误具体上下文与作为错误值类型的 error 接口类型的解耦，也体现了 Go 组合设计哲学中“正交”的理念。

到这里，我们已经基本了解了 Go 错误处理机制、统一的错误值类型，以及错误值构造方法。在这些基础上，我们可以再进一步，学习 Go 语言的几种错误处理的惯用策略，学习这些策略将有助于我们提升函数错误处理设计的能力。

策略一：透明错误处理策略

简单来说，Go 语言中的错误处理，就是根据函数 / 方法返回的 error 类型变量中携带的错误值信息做决策，并选择后续代码执行路径的过程。

这样，最简单的错误策略莫过于完全不关心返回错误值携带的具体上下文信息，只要发生错误就进入唯一的错误处理执行路径，比如下面这段代码：

复制代码

```
err := doSomething()
if err != nil {
    // 不关心err变量底层错误值所携带的具体上下文信息
    // 执行简单错误处理逻辑并返回
    ... ...
    return err
}
```

这也是 Go 语言中**最常见的错误处理策略**，80% 以上的 Go 错误处理情形都可以归类到这种策略下。在这种策略下，由于错误处理方并不关心错误值的上下文，所以错误值的构造方（如上面的函数`doSomething`）可以直接使用 Go 标准库提供的两个基本错误值构造方法`errors.New`和`fmt.Errorf`来构造错误值，就像下面这样：

复制代码

```
func doSomething(...) error {
    ... ...
    return errors.New("some error occurred")
}
```

这样构造出的错误值代表的上下文信息，对错误处理方是透明的，因此这种策略称为**“透明错误处理策略”**。在错误处理方不关心错误值上下文的前提下，透明错误处理策略能最大程度地减少错误处理方与错误值构造方之间的耦合关系。

策略二：“哨兵”错误处理策略

当错误处理方不能只根据“透明的错误值”就做出错误处理路径选取的情况下，错误处理方会尝试对返回的错误值进行检视，于是就有可能出现下面代码中的**反模式**：

复制代码

```
data, err := b.Peek(1)
if err != nil {
    switch err.Error() {
    case "bufio: negative count":
        // ... ...
        return
    case "bufio: buffer full":
        // ... ...
        return
    case "bufio: invalid use of UnreadByte":
        // ... ...
        return
    default:
        // ... ...
        return
    }
}
```

简单来说，反模式就是，错误处理方以透明错误值所能提供的唯一上下文信息（描述错误的字符串），作为错误处理路径选择的依据。但这种“反模式”会造成严重的**隐式耦合**。这也就意味着，错误值构造方不经意间的一次错误描述字符串的改动，都会造成错误处理方处理行为的变化，并且这种通过字符串比较的方式，对错误值进行检视的性能也很差。

那这有什么办法吗？Go 标准库采用了定义导出的（Exported）“哨兵”错误值的方式，来辅助错误处理方检视（inspect）错误值并做出错误处理分支的决策，比如下面的 bufio 包中定义的“哨兵错误”：

复制代码

```
// $GOROOT/src/bufio/bufio.go
var (
    ErrInvalidUnreadByte = errors.New("bufio: invalid use of UnreadByte")
    ErrInvalidUnreadRune = errors.New("bufio: invalid use of UnreadRune")
    ErrBufferFull        = errors.New("bufio: buffer full")
    ErrNegativeCount     = errors.New("bufio: negative count")
)
```

下面的代码片段利用了上面的哨兵错误，进行错误处理分支的决策：

复制代码

```
data, err := b.Peek(1)
if err != nil {
```

```
    switch err {
    case bufio.ErrNegativeCount:
        // ... ...
        return
    case bufio.ErrBufferFull:
        // ... ...
        return
    case bufio.ErrInvalidUnreadByte:
        // ... ...
        return
    default:
        // ... ...
        return
    }
}
```

你可以看到，一般“哨兵”错误值变量以 ErrXXX 格式命名。和透明错误策略相比，“哨兵”策略让错误处理方在有检视错误值的需求时候，可以“有的放矢”。

不过，对于 API 的开发者而言，暴露“哨兵”错误值也意味着这些错误值和包的公共函数 / 方法一起成为了 API 的一部分。一旦发布出去，开发者就要对它进行很好的维护。
而“哨兵”错误值也让使用这些值的错误处理方对它产生了依赖。

从 Go 1.13 版本开始，标准库 errors 包提供了 Is 函数用于错误处理方对错误值的检视。Is 函数类似于把一个 error 类型变量与“哨兵”错误值进行比较，比如下面代码：

复制代码

```
// 类似 if err == ErrOutOfBounds{ … }
if errors.Is(err, ErrOutOfBounds) {
    // 越界的错误处理
}
```

不同的是，如果 error 类型变量的底层错误值是一个包装错误（Wrapped Error），errors.Is 方法会沿着该包装错误所在错误链（Error Chain)，与链上所有被包装的错误（Wrapped Error）进行比较，直至找到一个匹配的错误为止。下面是 Is 函数应用的一个例子：

复制代码

```
var ErrSentinel = errors.New("the underlying sentinel error")

```

```
func main() {
  err1 := fmt.Errorf("wrap sentinel: %w", ErrSentinel)
  err2 := fmt.Errorf("wrap err1: %w", err1)
    println(err2 == ErrSentinel) //false
  if errors.Is(err2, ErrSentinel) {
    println("err2 is ErrSentinel")
    return
  }

  println("err2 is not ErrSentinel")
}
```

在这个例子中，我们通过 fmt.Errorf 函数，并且使用 %w 创建包装错误变量 err1 和 err2，其中 err1 实现了对 ErrSentinel 这个“哨兵错误值”的包装，而 err2 又对 err1 进行了包装，这样就形成了一条错误链。位于错误链最上层的是 err2，位于最底层的是 ErrSentinel。之后，我们再分别通过值比较和 errors.Is 这两种方法，判断 err2 与 ErrSentinel 的关系。运行上述代码，我们会看到如下结果：

复制代码

```
false
err2 is ErrSentinel
```

我们看到，通过比较操作符对 err2 与 ErrSentinel 进行比较后，我们发现这二者并不相同。而 errors.Is 函数则会沿着 err2 所在错误链，向下找到被包装到最底层的“哨兵”错误值`ErrSentinel`。

所以，如果你使用的是 Go 1.13 及后续版本，我建议你尽量使用`errors.Is`方法去检视某个错误值是否就是某个预期错误值，或者包装了某个特定的“哨兵”错误值。

策略三：错误值类型检视策略

上面我们看到，基于 Go 标准库提供的错误值构造方法构造的“哨兵”错误值，除了让错误处理方可以“有的放矢”的进行值比较之外，并没有提供其他有效的错误上下文信息。那如果遇到错误处理方需要错误值提供更多的“错误上下文”的情况，上面这些错误处理策略和错误值构造方式都无法满足。

这种情况下，我们需要通过自定义错误类型的构造错误值的方式，来提供更多的“错误上下文”信息。并且，由于错误值都通过 error 接口变量统一呈现，要得到底层错误类型携

带的错误上下文信息，错误处理方需要使用 Go 提供的**类型断言机制**（Type Assertion）或**类型选择机制**（Type Switch），这种错误处理方式，我称之为**错误值类型检视策略**。

我们来看一个标准库中的例子加深下理解，这个 json 包中自定义了一个`UnmarshalTypeError`的错误类型：

复制代码

```
// $GOROOT/src/encoding/json/decode.go
type UnmarshalTypeError struct {
    Value  string
    Type   reflect.Type
    Offset int64
    Struct string
    Field  string
}
```

错误处理方可以通过错误类型检视策略，获得更多错误值的错误上下文信息，下面就是利用这一策略的 json 包的一个方法的实现：

复制代码

```
// $GOROOT/src/encoding/json/decode.go
func (d *decodeState) addErrorContext(err error) error {
    if d.errorContext.Struct != nil || len(d.errorContext.FieldStack) > 0 {
        switch err := err.(type) {
        case *UnmarshalTypeError:
            err.Struct = d.errorContext.Struct.Name()
            err.Field = strings.Join(d.errorContext.FieldStack, ".")
            return err
        }
    }
    return err
}
```

我们看到，这段代码通过类型 switch 语句得到了 err 变量代表的动态类型和值，然后在匹配的 case 分支中利用错误上下文信息进行处理。

这里，一般自定义导出的错误类型以`XXXError`的形式命名。和“哨兵”错误处理策略一样，错误值类型检视策略，由于暴露了自定义的错误类型给错误处理方，因此这些错误类

型也和包的公共函数 / 方法一起，成为了 API 的一部分。一旦发布出去，开发者就要对它们进行很好的维护。而它们也让使用这些类型进行检视的错误处理方对其产生了依赖。

从 Go 1.13 版本开始，标准库 errors 包提供了As函数给错误处理方检视错误值。As函数类似于通过类型断言判断一个 error 类型变量是否为特定的自定义错误类型，如下面代码所示：

复制代码

```
// 类似 if e, ok := err.(*MyError); ok { … }
var e *MyError
if errors.As(err, &e) {
    // 如果err类型为*MyError，变量e将被设置为对应的错误值
}
```

不同的是，如果 error 类型变量的动态错误值是一个包装错误，errors.As函数会沿着该包装错误所在错误链，与链上所有被包装的错误的类型进行比较，直至找到一个匹配的错误类型，就像 errors.Is 函数那样。下面是As函数应用的一个例子：

复制代码

```
type MyError struct {
    e string
}

func (e *MyError) Error() string {
    return e.e
}

func main() {
    var err = &MyError{"MyError error demo"}
    err1 := fmt.Errorf("wrap err: %w", err)
    err2 := fmt.Errorf("wrap err1: %w", err1)
    var e *MyError
    if errors.As(err2, &e) {
        println("MyError is on the chain of err2")
        println(e == err)
        return
    }
    println("MyError is not on the chain of err2")
}
```

运行上述代码会得到：

复制代码

```
MyError is on the chain of err2
true
```

我们看到，errors.As函数沿着 err2 所在错误链向下找到了被包装到最深处的错误值，并将 err2 与其类型 * MyError成功匹配。匹配成功后，errors.As 会将匹配到的错误值存储到 As 函数的第二个参数中，这也是为什么println(e == err)输出 true 的原因。

所以，如果你使用的是 Go 1.13 及后续版本，请尽量使用errors.As方法去检视某个错误值是否是某自定义错误类型的实例。

策略四：错误行为特征检视策略

不知道你注意到没有，在前面我们已经讲过的三种策略中，其实只有第一种策略，也就是“透明错误处理策略”，有效降低了错误的构造方与错误处理方两者之间的耦合。虽然前面的策略二和策略三，都是我们实际编码中有效的错误处理策略，但其实使用这两种策略的代码，依然在错误的构造方与错误处理方两者之间建立了耦合。

那么除了“透明错误处理策略”外，我们是否还有手段可以降低错误处理方与错误值构造方的耦合呢？

在 Go 标准库中，我们发现了这样一种错误处理方式：**将某个包中的错误类型归类，统一提取出一些公共的错误行为特征，并将这些错误行为特征放入一个公开的接口类型中**。这种方式也被叫做错误行为特征检视策略。

以标准库中的net包为例，它将包内的所有错误类型的公共行为特征抽象并放入net.Error这个接口中，如下面代码：

复制代码

```
// $GOROOT/src/net/net.go
type Error interface {
    error
    Timeout() bool
    Temporary() bool
}
```

我们看到，net.Error 接口包含两个用于判断错误行为特征的方法：Timeout 用来判断是否是超时（Timeout）错误，Temporary 用于判断是否是临时（Temporary）错误。

而错误处理方只需要依赖这个公共接口，就可以检视具体错误值的错误行为特征信息，并根据这些信息做出后续错误处理分支选择的决策。

这里，我们再看一个 http 包使用错误行为特征检视策略进行错误处理的例子，加深下理解：

复制代码

```
// $GOROOT/src/net/http/server.go
func (srv *Server) Serve(l net.Listener) error {
    ... ...
    for {
        rw, e := l.Accept()
        if e != nil {
            select {
            case <-srv.getDoneChan():
                return ErrServerClosed
            default:
            }
            if ne, ok := e.(net.Error); ok && ne.Temporary() {
                // 注：这里对临时性(temporary)错误进行处理
                ... ...
                time.Sleep(tempDelay)
                continue
            }
            return e
        }
        ...
    }
    ... ...
}
```

在上面代码中，Accept 方法实际上返回的错误类型为*OpError，它是 net 包中的一个自定义错误类型，它实现了错误公共特征接口net.Error，如下代码所示：

复制代码

```
// $GOROOT/src/net/net.go
type OpError struct {
    ... ...
    // Err is the error that occurred during the operation.
    Err error
```

```
}

type temporary interface {
    Temporary() bool
}

func (e *OpError) Temporary() bool {
  if ne, ok := e.Err.(*os.SyscallError); ok {
      t, ok := ne.Err.(temporary)
      return ok && t.Temporary()
  }
  t, ok := e.Err.(temporary)
  return ok && t.Temporary()
}
```

因此，OpError 实例可以被错误处理方通过net.Error接口的方法，判断它的行为是否满足 Temporary 或 Timeout 特征。

小结

好了，今天的课讲到这里就结束了。在这一讲中，我们重点讲解了 Go 函数设计中的一个重要环节：**错误处理设计**，希望通过这节课的内容，能帮助你建立起代码设计的意识，提高函数设计的水平。

Go 语言继承了 C 语言的基于值比较的错误处理机制，但又在 C 语言的基础上做出了优化，也就是说，Go 函数通过支持多返回值，消除了 C 语言中将错误状态值与返回给函数调用者的信息耦合在一起的弊端。

Go 语言还统一错误类型为 error 接口类型，并提供了多种快速构建可赋值给 error 类型的错误值的函数，包括 errors.New、fmt.Errorf 等，我们还讲解了使用统一 error 作为错误类型的优点，你要深刻理解这一点。

基于 Go 错误处理机制、统一的错误值类型以及错误值构造方法的基础上，Go 语言形成了多种错误处理的惯用策略，包括透明错误处理策略、“哨兵”错误处理策略、错误值类型检视策略以及错误行为特征检视策略等。这些策略都有适用的场合，但**没有某种单一的错误处理策略可以适合所有项目或所有场合**。

在错误处理策略选择上，我有一些个人的建议，你可以参考一下：

请尽量使用“透明错误”处理策略，降低错误处理方与错误值构造方之间的耦合；

如果可以通过错误值类型的特征进行错误检视，那么请尽量使用“错误行为特征检视策略”；

在上述两种策略无法实施的情况下，再使用“哨兵”策略和“错误值类型检视”策略；

Go 1.13 及后续版本中，尽量用`errors.Is`和`errors.As`函数替换原先的错误检视比较语句。

思考题

这节课，我们列出了一些惯用的错误处理策略，当然，Go 社区关于错误处理策略的讨论可能不止这些，你还见过哪些比较实用的错误处理策略吗？不妨在留言区和我们探讨一下吧。

欢迎你把这节课分享给更多对 Go 语言的错误处理机制感兴趣的朋友。我是 Tony Bai，我们下节课见。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 5 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 21 | 函数：请叫我“一等公民”

下一篇 期中测试 | 一起检验下你的学习成果吧

更多学习推荐

2021 最新大厂
Go 工程师面试真题

大厂面试真题 + 金九银十全新整理 + 核心知识全覆盖

免费领取

精选留言 (8)

写留言

在下宝龙、
2021-12-01

最后一部分的策略四的设计不是很懂 。。。

展开

共 5 条评论 > 4

aoe
2021-12-02

多值返回的错误处理很好用。知道了4种错误使用场景，变强了！

1

Geralt of Rivia
2021-12-02

忽略错误算不算实用的策略呢

展开

1

i6u
2021-12-01

if err != nil { return err; }

看错误怎么能少得了这个

展开

共 1 条评论 1

Summer 空城

2021-12-01

老师，何时使用方法，何时使用函数，指点下呗

展开

1

罗杰

2021-12-01

再读错误处理，依然收获满满。

展开

1

qinsi

2021-12-04

errors.Wrap多好用

展开

功夫熊猫

2021-12-03

if err != nil{

}